Algoritmos y Estructura de Datos: Examen 2 (Solución)

Grados Ing. Inf. y Mat. Inf. Noviembre 2011

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Apellidos: .....

Nombre: .....

DNI / NIE: ..... **Núm. matrícula:** .....

Normas

- Este examen consta de **6 preguntas** en **6 páginas**.
- La puntuación total del examen es de **10 puntos**.
- La duración total del examen es de **90 minutos**.
- El examen debe contestarse **en las hojas que se proporcionan**.
- Deben rellenarse los campos obligatorios **apellidos, nombre, y DNI/NIE**.
- Las calificaciones provisionales de este examen se publicarán en el Aula Virtual el **14 de diciembre** junto con las soluciones. La fecha de la revisión de este examen es el **16 de diciembre**. La hora y lugar de dicha revisión se anunciará en el Aula Virtual.
- En las preguntas con varias opciones, **sólo hay una respuesta válida por pregunta**. En este caso toda pregunta en que se marque más de una respuesta se considerará incorrectamente contestada y toda pregunta incorrectamente contestada restará del examen una cantidad de puntos igual a la puntuación de la pregunta dividido por el número de alternativas ofrecidas en la misma menos uno. Es decir, una respuesta incorrecta en una pregunta de un punto con cuatro alternativas resta $\frac{1}{3}$ de punto.

(1 punto) 1. Se tiene un árbol binario perfecto (todos los nodos internos tienen dos hijos) que almacena caracteres en los nodos. El recorrido del árbol en inorden produce la siguiente secuencia de elementos:

B D E A F G C

Se pide: Indicar cuál de entre las siguientes secuencias de elementos es la que produce el recorrido en postorden de dicho árbol:

(a) A B C D E F G

(b) C E D F G E A

(c) B E D F C G A ✔

(d) C G F A E D B

(1 punto) 2. Se tiene un método que toma como parámetro un vector o «array» de elementos y devuelve un objeto lista de clase `NodePositionList` con los elementos del vector ordenados de forma creciente. **Se pide:** Indicar la complejidad en el caso peor de dicho método, razonando la respuesta.

La inserción en una lista ordenada tiene complejidad lineal. Para cada índice i de 0 hasta n (tamaño del vector) se inserta el elemento $v[i]$ en la lista de tamaño i de forma ordenada. La complejidad es la suma de las complejidades de las inserciones: $1 + 2 + \ldots + n = (n^2 - n)/2$ que es $O(n^2)$.

(1 punto) 3. **Se pide:** Indicar la altura máxima y la altura mínima que puede tener un árbol binario del que solamente se sabe que tiene n nodos. Razone la respuesta.

La altura máxima es $n-1$, que se da si todos los nodos internos del árbol tienen únicamente un hijo. La altura mínima es $\log n$, que se da si todos los nodos internos del árbol tienen dos hijos, exceptuando los del penúltimo nivel que pueden tener sólo un hijo.

(2 puntos) 4. Se desea añadir el método `subset` a la clase `IterableBitVectSet` utilizada en las clases de laboratorio de la asignatura para implementar conjuntos iterables:

```
public classs IterableBitVectSet<E extends Enum<E>> extends BitVectSet<E>
  implements Set<E>, Iterable<E> {
   ...
   public boolean subset(IterableBitVectSet<E> s1, IterableBitVectSet<E> s2)
     throws InvalidSetException {
     /** COMPLETAR **/
   }
```

Se pide: Escribir el código Java del método `subset` que indica si el conjunto iterable `s1` es un subconjunto del conjunto iterable `s2`, es decir, si todos los elementos de `s1` están en `s2`. Recordamos que el conjunto vacío es un subconjunto de cualquier conjunto (y por tanto también es subconjunto de sí mismo).

```
public boolean subset(IterableBitVectSet<E> s1, IterableBitVectSet<E> s2)
  throws InvalidSetException {
    if (s1 == null || s2 == null) throw new InvalidSetException();
    if (s1.isEmpty()) return true;
    else {
      boolean esta;
      Iterator<E> it1 = s1.iterator();
      while (it1.hasNext() && (esta = s2.member(it1.next()))) ;
      return esta;
    }
  }
}
```

(3 puntos) 5. Se desea añadir el método `negar` a la clase `BitVectSet<E extends Enum<E>>` que implementa conjuntos acotados mediante vectores de bits:

```
public class BitVectSet<E extends Enum<E>> implements Set<E> {
  ...
  public void negar() { /** COMPLETAR **/ }
}
```

Dicho método realiza el complemento del conjunto cuando el conjunto no es vacío. Si el conjunto es vacío el método no tiene efecto. El complemento de un conjunto `s` de enumerados es el conjunto formado por todos los elementos del enumerado que no están en `s`. Mostramos un caso de uso:

```
enum Dias {L, M, X, J, V, S, D};
BitVectSet<Dias> s = new BitVectSet<Dias>();
s.add(Dias.L);
s.add(Dias.V);
s.negate(); // Contiene Dias.M, Dias.X, Dias.J, Dias.S y Dias.D
```

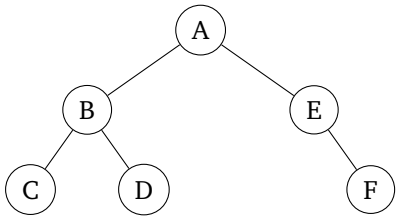
Es un método de la clase y por tanto tiene acceso a los atributos de la misma. Además no se aportan interfaces.

```
public void negar() {
  if (size > 0) {
    for (int i = 0; i < bv.length; i++)
      bv[i] = ! bv[i];
    size = bv.length - size;
  }
}
```

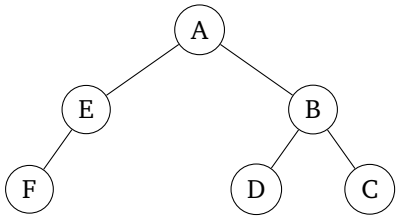
(2 puntos) 6. El espejo de un nodo `p` se define de la siguiente forma:

- Si `p` es un nodo externo su espejo es él mismo.
- Si `p` es un nodo interno, entonces:
  - Si `p` tiene un nodo hijo izquierdo `i`, entonces el nodo espejo de `p` tiene como hijo derecho el nodo espejo de `i`.
  - Si `p` tiene un nodo hijo derecho `d`, entonces el nodo espejo de `p` tiene como hijo izquierdo el nodo espejo de `d`.

Por ejemplo, dado el siguiente árbol binario:



el nodo espejo de su nodo raíz es el nodo raíz del siguiente árbol, el cual se obtiene visualmente dando la vuelta a esta hoja del examen, viéndose dicho árbol como a través de un espejo:



Se pide: Implementar en Java el método

```
public BTPosition<E> espejo(BinaryTree<E> t, BTPosition<E> p)
```

que toma un árbol binario `t` y un nodo de dicho árbol `p` y debe modificar `p` con su nodo espejo y también devolverlo como resultado.

El interfaz `BinaryTree<E>` está disponible en el Apéndice A.2 (página 5), el interfaz `BTPosition<E>` está disponible en el Apéndice A.1 (página 5), y el interfaz `Tree<E>` está disponible en el Apéndice A.3 (página 6).

No se aporta ninguna clase que implemente nodos del árbol con lo que la solución debe utilizar métodos de los interfaces proporcionados.

Solución «bottom-up» que computa el espejo de los hijos del nodo y después los intercambia:

```
public BTPosition<E> espejo(BinaryTree<E> t, BTPosition<E> p) {
  if (p!=null && t.isInternal(p)) {
    BTPosition<E> ei = espejo(t,p.getLeft());
    BTPosition<E> ed = espejo(t,p.getRight());
    p.setRight(ei);
    p.setLeft(ed);
  }
  return p;
}
```

Solución «top-down» que intercambia los hijos y después computa el espejo de éstos:

```
 public BTPosition<E> espejo(BinaryTree<E> t, BTPosition<E> p) {
   if (p!= null && t.isInternal(p)) {
     BTPosition<E> i = p.getLeft();
     BTPosition<E> r = p.getRight();
     p.setLeft(r);
     p.setRight(l);
     espejo(t,l) ; /* modifica l */
     espejo(t,r) ; /* modifica r */
  }
  return p;
}
```

Solución «bottom-up» que lanza `InvalidPositionException` si `p` es null. Las llamadas recursivas no lanzan la excepción:

```
public BTPosition<E> espejo(BinaryTree<E> t, BTPosition<E> p) {
  BTPosition<E> ei,ed;  /* inicialmente null */
  if (t.isInternal(p)) {  /* excepción si p null */
    if (t.hasLeft(p))  ei = espejo(t,p.getLeft());
    if (t.hasRight(p)) ed = espejo(t,p.getRight());
    p.setRight(ei);
    p.setLeft(ed);
  }
  return p;
}
```

Solución «top-down» que lanza `InvalidPositionException` si `p` es null. Las llamadas recursivas no lanzan la excepción:

```
 public BTPosition<E> espejo(BinaryTree<E> t, BTPosition<E> p) {
   if (t.isInternal(p)) {
     BTPosition<E> i = p.getLeft();
     BTPosition<E> r = p.getRight();
     p.setLeft(r);
     p.setRight(l);
     if (l!=null) espejo(t,l) ; /* modifica l */
     if (r!=null) espejo(t,r) ; /* modifica r */
  }
  return p;
}
```

A. Código de apoyo

A.1. Interfaz `BTPosition<E>`

```
package net.datastructures;
public interface BTPosition<E> extends Position<E> { // inherits element()
  public void setElement(E o);
  public BTPosition<E> getLeft();
  public void setLeft(BTPosition<E> v);
  public BTPosition<E> getRight();
  public void setRight(BTPosition<E> v);
  public BTPosition<E> getParent();
  public void setParent(BTPosition<E> v);
}
```

A.2. Interfaz `BinaryTree<E>`

```
public interface BinaryTree<E> extends Tree<E> {
  /** Returns the left child of a node. */
  public Position<E> left(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns the right child of a node. */
  public Position<E> right(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns whether a node has a left child. */
  public boolean hasLeft(Position<E> v) throws InvalidPositionException;

  /** Returns whether a node has a right child. */
  public boolean hasRight(Position<E> v) throws InvalidPositionException;
}
```

A.3. Interfaz `Tree<E>`

```
package net.datastructures;
import java.util.Iterator;

public interface Tree<E> {
  /** Returns the number of nodes in the tree. */
  public int size();

  /** Returns whether the tree is empty. */
  public boolean isEmpty();

  /** Returns an iterator of the elements stored in the tree. */
  public Iterator<E> iterator();

  /** Returns an iterable collection of the the nodes. */
  public Iterable<Position<E>> positions();

  /** Replaces the element stored at a given node. */
  public E replace(Position<E> v, E e)
    throws InvalidPositionException;

  /** Returns the root of the tree. */
  public Position<E> root() throws EmptyTreeException;

  /** Returns the parent of a given node. */
  public Position<E> parent(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns an iterable collection of the children of a given node. */
  public Iterable<Position<E>> children(Position<E> v)
    throws InvalidPositionException;

  /** Returns whether a given node is internal. */
  public boolean isInternal(Position<E> v)
    throws InvalidPositionException;

  /** Returns whether a given node is external. */
  public boolean isExternal(Position<E> v)
    throws InvalidPositionException;

  /** Returns whether a given node is the root of the tree. */
  public boolean isRoot(Position<E> v)
    throws InvalidPositionException;
}
```